Polynomial Exact-3-SAT Solving Algorithm

Louis Coder
louis@louis-coder.com

December 2012

Abstract

In this document I want to introduce and explain an algorithm that determines the solvability state (solvable or unsatisfiable) of any exact-3-SAT formula in polynomial time. It is for sure that the algorithm has polynomial runtime, even in the worst case, as the runtime is artificially limited. The question is only if the algorithm does always output the correct result. I suppose it does, due to a proof of correctness that will be shown in this document, and the evidence that an implementation of the algorithm solved millions of test formulae without any error. Furthermore this document provides a download link to a (Windows) demo solver program that you can try out. If the algorithm should really be correct, it would solve the P-NP problem by proving that P = NP [1].

1. Introduction

The P-NP-problem is a still unsolved problem well known by computer scientists. The problem asks if it is possible to solve those problems in polynomial time that are currently associated to the complexity class 'NP'. One of those problems that lies in NP is exact-3-SAT.

Exact-3-SAT is a variant of the such-called satisfiability problem. The task is to determine if there's a solution for a formula given as conjunctive normal form with exactly 3 different literals per clause [2].

Example:

$$Exact-3-SAT-CNF=(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3)$$

The AND-terms are called clauses, the OR-terms within a clause are called literals [3]. A solution would be e.g. a table that assigns either true or false to every literal.

One possible solution for the just shown CNF would be:

$x_1 = true$

$x_2 = true$

$x_3 = true$

…because when we insert this into the CNF we get:

$$Exact-3-SAT-CNF=(true \vee true \vee true) \wedge (true \vee false \vee false) \wedge (false \vee true \vee true)$$

As in every clause there's at least once true, the CNF itself becomes true. It might be that the CNF can never become true, no matter how you choose the values of the literals. My algorithm shall determine the solvability state of any given CNF, that means whether it is possible at all to get the whole CNF true.

Note that in the following, 'algorithm' means the polynomial algorithm this document concerns about. Furthermore the expressions 'clause' and 'AND-term' will be used synonymously within this document.

2. Algorithm

The following algorithm to solve the satisfiability problem was developed by myself. I do not know if there are already similar algorithms existing. I did not (knowingly) extend or change any existing algorithm that is described in literature. I just wrote down unsatisfiable CNFs on a sheet of paper and tried to invent rules how you can determine the CNFs‘ solvability states. If an algorithm worked on paper, I implemented it as a program and did a lot of tests. After 2 years proceeding like this, I found the following algorithm, of which I think it is correct. The references to literature have mainly been sought at a time the algorithm was already finished.

2.1 Main Idea

The most important idea of the algorithm is to do the ‘clause overlaying’, as I call it. You can ‘overlay’ two source clauses to the ‘overlaid clause’ if the source clauses contain exactly one literal once negated and once not negated . This literal does not appear in the overlaid clause any more, all other literals are just copied to the overlaid clause.

Example:

$$(x_1 \vee x_2 \vee x_3) overlaid\_over (x_1 \vee \overline{x_2} \vee \overline{x_4}) = (x_1 \vee x_3 \vee \overline{x_4})$$

It might be that this ‘clause overlaying’ is already known as ‘resolution’ [4]. As I am not absolutely sure that these two concepts are really equivalent, I keep on speaking about ‘overlaying’ (it was my own invention to use this expression).

To get the solvability state you just overlay all possible pairs of clauses of the given CNF. An overlaid clause can participate in an overlaying operation again.

One important detail is the following: you try to get overlaid clauses with as few literals as possible. So, for instance, add an overlaid clause with 3 literal to the CNF only if there was none creatable with 2 or 1 literal(s).

Furthermore, do not add an overlaid clause if it is a sub set of any existing AND-term. For instance, do not add

$$(\overline{x_1} \vee x_2 \vee x_3)$$

if there’s

$$(\overline{x_1} \vee x_2)$$

existing in the CNF. A clause S is a sub set of a clause L if S contains at least all literals of L (negation states in S, i.e. if literals are negated or not, must match the ones in L, of course).

You do the overlaying until one of the following three cases happens:

- The empty clause with no literals appears => the CNF is unsatisfiable, there is no solution existing
- You cannot create new overlaid clauses any more => the CNF is solvable, it exists some solution
- You exceeded a special limit of clauses (will be explained later) => the CNF is solvable

2.2 Example Usage

In the following example, I demonstrate how we find out that the given CNF is unsatisfiable. To keep the example short and easy to understand, I use an exact-2-SAT formula, with 2 instead of 3 literals in the initial CNF’s AND-terms. The proceeding with 3 literals would be equivalent, except that it would take more AND-terms to create.

Step 0: the given CNF (‘initial CNF‘)
$(x_1 \vee x_2) \wedge (\overline{x_1} \vee x_2) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_2})$
Step 1: overlay first and second clause
$(x_1 \vee x_2) \wedge (\overline{x_1} \vee x_2) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (x_2)$
Step 2: overlay third and fourth clause
$(x_1 \vee x_2) \wedge (\overline{x_1} \vee x_2) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (x_2) \wedge (\overline{x_2})$
Step 3: overlay fifth and sixth clause
$(x_1 \vee x_2) \wedge (\overline{x_1} \vee x_2) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (x_2) \wedge (\overline{x_2}) \wedge ( )$

3. Pseudo Code

3.1 The Algorithm in Pseudo Code

The following is a Basic-like pseudo code listing of the demo solver you can download. It implements the polynomial exact-3-SAT solving algorithm.

```
ReDo:    // goto-target (label)                                    // 1)
For OverlaidClauseLiteralCountMax = 0 To CNF.MaxLiteralIndex       // 2)
    For OuterLoop = 1 To CNF.ClauseNumber                          // 3)
        For InnerLoop = OuterLoop + 1 To CNF.ClauseNumber          // 4)
            OverlaidClause = Overlay(
                CNF.Clause[OuterLoop],
                CNF.Clause[InnerLoop])                             // 5)

            If OverlaidClause.LiteralCount > OverlaidClauseLiteralCountMax Or
               OverlaidClause.LiteralCount < 0 Then                // 6)
                Next InnerLoop
            EndIf

            If OverlaidClause.LiteralCount == 0 Then               // 7)
                Return Unsatisfiable
            EndIf

            If IsSubsetOfAnyClauseInCNF(OverlaidClause, CNF) Then
                Next InnerLoop                                     // 8)
            EndIf

            CNF.AddClause(OverlaidClause)                          // 9)

            If WorkingCNF.ClauseNumber > ExpectedClauseCountMax Then
                Return Solvable                                    // 10)
            EndIf

            Goto ReDo                                              // 11)
        Next InnerLoop
    Next OuterLoop
Next OverlaidClauseLiteralCountMax

Return Solvable                                                    // 12)
```

3.2 Explanation of the Pseudo Code

The following numbers refer to the numbers in the pseudo code.

1. we jump to this code location every time we added an overlaid clause to the CNF.
2. `OverlaidClauseLiteralCountMax` is the maximal literal count an overlaid clause can have to be added to the working CNF. IMPORTANT: we try to add overlaid clauses that have as few literals as possible to reduce the number of clauses in the CNF (the less literals a clause has, the less possible clauses you can create out of these literals).
3. The index of the first clause that will take part in an overlaying operation.

4. The index of the second clause that will take part in an overlaying operation. As it does not matter in which order we overlay the clauses, we can start ‘behind’ the index of the first clause. Note that, in the original C++ solver code, we additionally use an array to not overlay pairs that have been overlaid once before (the array is named ‘Start[][]’).
5. We call a procedure that does the overlaying.
6. If the overlaid clause has too many literals (see point 2)) or the overlaying operation could not be done as there is not exactly one conflict existing between the two clauses to overlay (the overlaying procedure then could set the literal count to -1, for example), we continue with the next clause pair.
7. If we get the empty clause, the CNF is unsatisfiable.
8. If there’s any clause existing in the CNF that contains the current overlaid clause, that means if the overlaid clause is a sub set of any existing clause, throw away the overlaid clause (i.e. ignore it).
9. The overlaid clause is added to the CNF and may take part in another overlaying operation.
10. If the number of clauses in the CNF exceeds a special limit (`ExpectedClauseCountMax` was pre-calculated before) we can suppose the CNF is solvable.
11. Every time we added an overlaid clause to the CNF, we jump to the beginning, which mainly results in a reset of `OverlaidClauseLiteralCountMax`, so that we add clauses with as few literals as possible.
12. If we reach this code line, no new overlaid clause was created any more.

4. Complexity

In the following paragraphs, clause S shall be called a ‚shorter clause‘ than a clause L, if S has less literals than L. A ‚longer‘ clause is one with more literals. An x-literal clause shall be one with x literals. These expressions were introduced by myself.

Also we must define what ‚complexity‘ shall be. I defined the complexity as the total number of clauses (respectively AND-terms, synonymous) added to the working CNF until the solver successfully determined the solvability state.

4.1 Worst-Case Complexity of the Algorithm

I say, the total number of clauses ever added to the CNF will, if the CNF is unsolvable, never exceed the count of all possible clauses with 3 or less literals. ‚Possible clauses‘ means all clauses that are creatable at all (containing the mentioned number of literals).

What is the count of all possible clauses with 3 or less literals?

One could say, first the total number of clauses with 3 literals and highest possible literal index = `IndexMax` would be:

$$(IndexMax)*(IndexMax-1)*(IndexMax-2)*8$$

…as we have `IndexMax` possibilities to choose the first literal index, one less for the second one and two less for the third one. Furthermore we have 8 possible true/false combinations:

Literal 1	Literal 2	Literal 3
false	false	false
false	false	true
false	true	false
false	true	true
true	false	false
true	false	true
true	true	false
true	true	true

But, this calculation is not true as this would also contain vice-versa (swapped) combinations, like
$(x_1 \vee x_2 \vee x_3)$
and
$(x_2 \vee x_1 \vee x_3)$
(as the order of literals is not important, these two AND-terms are seen as identical).

So here's the correct calculation of the count of clauses with 3 literals and highest literal index = `IndexMax`:

$$\sum_{i=1}^{IndexMax-2} \left(\sum_{j=i+1}^{IndexMax-1} \left(\sum_{k=j+1}^{IndexMax} 8 \right) \right)$$

For the total complexity you must add the count of clauses with 2 or 1 literal(s). Finally the exact worst-case complexity of the polynomial solver is:

$$\sum_{i=1}^{IndexMax-2} \left(\sum_{j=i+1}^{IndexMax-1} \left(\sum_{k=j+1}^{IndexMax} 8 \right) \right) + \left(\sum_{j=1}^{IndexMax-1} \left(\sum_{k=j+1}^{IndexMax} 4 \right) \right) + \sum_{k=1}^{IndexMax} 2$$

This means, our solver has a complexity of $O(n^3)$.

The origin of these formulas are my own considerations. They are rather easy to understand and I could not falsify them in tests, so it is very probable that the formulas are correct.

4.2 Reason for Worst-Case Complexity

The provided solver is programmed to output 'solvable' as soon as the CNF clause count exceeds the sum of all possible clauses with 3 or less literals (see 4.1). This seems to be logical, as in the initial CNF there are only clauses with exactly 3 literals, and most of them are overlaid to clauses with 2 or less literals.

But, there's a problem: it does happen that there are clauses with more than 3 literals created through overlaying, sometimes even ones with 4, 5 or more literals. In my tests this happened for both satisfiable and unsatisfiable CNFs.

The problem with more than 3 literals is the following:

- The calculated limit is the count of all clauses with 3 or less literals
- If we now add a clause with 4 literals, the maximal calculated clause count (see 4.1) could be exceeded, as the limit does not include the count of additional AND-terms with more than 3 literals
- The question is if we could count a CNF falsely as solvable as the expected clause count is reached, but we would get the empty clause if we kept on overlaying?

I say, even if clauses with more than 3 literals appear in an unsatisfiable CNF, the calculated limit is not exceeded.

Reason:

when a clause with more than 3 literals is added, all included clauses must be missing.

So if we add for example:
$(x_1 \vee x_2 \vee x_3 \vee x_4)$
then there must be missing:
$(x_1 \vee x_2 \vee x_3)$
$(x_1 \vee x_2 \vee x_4)$
$(x_1 \vee x_3 \vee x_4)$
$(x_2 \vee x_3 \vee x_4)$
$(x_1 \vee x_2)$
$(x_1 \vee x_3)$
$(x_1 \vee x_4)$
$(x_2 \vee x_3)$
$(x_2 \vee x_4)$
$(x_3 \vee x_4)$
(x_1)
(x_2)
(x_3)
(x_4)

The code line 8) in '3.1 […] Pseudo Code' would avoid that $(x_1 \vee x_2 \vee x_3 \vee x_4)$ (in the following called 'surplus clause') is added if there was at least one of the shorter clauses existing in the CNF.

But, what if all those shorter clauses are added afterwards, i.e. first the clause with 4 literals and then all shorter ones? Then we would have one AND-term more in the CNF than calculated, namely additional the 4-literal clause.

Even if this would happen, it is for sure that there are further AND-terms missing that were recognized by the limit calculated in 4.1. Because, imagine the surplus clause would have been created through the following overlaying operation:

$$(x_1 \vee x_2 \vee x_5) overlaid\_over(x_3 \vee x_4 \vee \overline{x_5}) = (x_1 \vee x_2 \vee x_3 \vee x_4)$$

That means that the two source clauses would have been existing in the CNF and they created the surplus clause.

But there must those clauses be missing that would have avoided the creation of the surplus one!

So if e.g.

$$(x_1 \vee x_2 \vee \overline{x_5})$$

would have existed in the initial CNF, the algorithm would have overlaid before (!) the surplus 4-literal clause would have been created:

$$(x_1 \vee x_2 \vee x_5) overlaid\_over(x_1 \vee x_2 \vee \overline{x_5}) = (x_1 \vee x_2)$$

So we would have gotten one of those clauses that are a superset of the surplus one.

That means, as soon as one 4-literal clause is added, there must at least one 3-literal clause be missing. Actually there are much more 3-literal clauses missing, namely all those that would create the 2-literal clauses listed farer above. So we will never exceed the calculated limit.

Note that I successfully determined the solvability state of millions of CNFs with the sample implementation (see 5.), this lets me suppose the just shown proof is correct. It is very probable that, if the algorithm would not be correct, there would have appeared at least one error.

5. Sample Implementation

I recommend you to download and try out the sample implementation - a Windows console program that allows you to solve random CNFs with the polynomial solver and compare the results with those of an exponential, fail-safe one. In the downloadable zip-file there's also the source code, which might be compiled under Linux, too (with some little code changes). In the zip-file there's also this document (v3.1) and an older version (v2.1) that explains how to use the sample implementation (see 'Documentation' directory).

http://www.louis-coder.com/Polynomial_3-SAT_Solver/Polynomial_3-SAT_Solver.zip

6. Proofs

6.1. Overlaying leads to Empty Clause

In the following paragraphs I want to prove that, if we can not (!) get the empty clause by overlaying, the CNF is solvable.

The CNF is unsatisfiable if there are at least two clauses that cannot both be satisfied by a solution.

For instance,
(x_1)
and
$(\overline{x_1})$
cannot both be satisfied at the same time.

Remember: overlaying is done with pairs of clauses. If these two clauses contain exactly one conflict literal (negated in the one clause and not negated in the other one) we can remove the conflict literal and write the resting literals into the overlaid clause, which has always fewer literals than those of the two source clauses (as the conflict literals have been removed).

If we can not overlay two AND-terms, it is for sure that there is a solution existing that satisfies both of them.

I want to prove this by just listing such cases where we cannot overlay and corresponding possible solutions. The first two lines are the two source clauses that are to be overlaid. Right after, there are possible solutions shown that satisfy both source clauses.

1.
$(x_1 \vee x_2)$
$(x_1 \vee x_2)$
Solution:
$x_1 = true$
or
$x_2 = true$

2.

$(\overline{x_1} \vee \overline{x_2})$

$(\overline{x_1} \vee \overline{x_2})$

Solution:

$x_1 = false$

or

$x_2 = false$

3.

$(\overline{x_1} \vee x_2)$

$(\overline{x_1} \vee x_2)$

Solution:

$x_1 = false$

or

$x_2 = true$

4.

$(\overline{x_1} \vee x_2)$

$(x_1 \vee \overline{x_2})$

Solution:

$x_1 = false \wedge x_2 = false$

or

$x_1 = true \wedge x_2 = true$

We see, if we cannot overlay at all there are only clauses left in the CNF that can all (!) be satisfied and thus the whole CNF has some solution!

Note that the listed solutions do also satisfy some AND-terms with more than 2 literals. The solutions for those clauses can be found similarly.

If there are clauses left with 1 literal and we cannot overlay, then these ones can be satisfied, too.
The only 1-literal clauses that cannot be overlaid are:

5.

x_1

x_1

Solution:

$x_1 = true$

6.

$\overline{x_1}$

$\overline{x_1}$

Solution:

$x_1 = false$

6.2. Sub Set Check

Is it valid to not add an overlaid clause if it is a sub set of any clause already existing in the CNF?

Remember that we do not add e.g.
$(x_1 \vee x_2 \vee x_3)$
if there's already
$(x_1 \vee x_2)$
existing.

I say this is valid because: to make the whole CNF solvable, all AND-terms must be satisfied by a solution.

That means
$(x_1 \vee x_2 \vee x_3)$
and also
$(x_1 \vee x_2)$
must be satisfied.

Imagine that
$(x_1 \vee x_2)$
is already existing in the CNF.

I say, if this clause can be satisfied, there's no need to add
$(x_1 \vee x_2 \vee x_3)$
as
$(x_1 \vee x_2) = true$
means the existing clause can be satisfied. If we insert this into the longer one, we get:
$((true) \vee x_3)$
…and this is always true.

So if
$(x_1 \vee x_2)$
can not (!) be satisfied, the whole CNF is unsatisfiable as all (!) clauses must be satisfied.

But if
$(x_1 \vee x_2)$
can be satisfied,
$(x_1 \vee x_2 \vee x_3)$
is also true in any case and we don’t need to add it to the CNF.

7. Acknowledgements

I thank the reader of this document for his/her interest and I would be even more happy if he/she would comment the algorithm, if possible. Also thanks to arXiv.org for publishing my article and the professors from University Erlangen-Nuremberg for educating me.

8. References

[1]:
1) http://de.wikipedia.org/wiki/P-NP-Problem (all Wikipedia versions from 2008-06-20)
2) lecture “Berechenbarkeit und formale Sprachen” by professor Wanka, winter term 2009/2010, University Erlangen-Nuremberg

[2]:
1) http://de.wikipedia.org/wiki/3-SAT (all Wikipedia versions from 2008-06-20)
2) lecture “Berechenbarkeit und formale Sprachen” by professor Wanka, winter term 2009/2010, University Erlangen-Nuremberg

[3]:
1) http://de.wikipedia.org/wiki/Konjunktive_Normalform (all Wikipedia versions from 2008-06-20)
2) lecture “Berechenbarkeit und formale Sprachen” by professor Wanka, winter term 2009/2010, University Erlangen-Nuremberg

[4]: http://de.wikipedia.org/wiki/Resolution_%28Logik%29 (all Wikipedia versions from 2008-06-20)